UCRL-94432
PREPRINT

PARALLEL COMPUTING AND MULTITASKING

David V. Anderson
Eric J. Horowitz
Alice E. Koniges
Michel G. McCoy

This paper was prepared for submittal to
Computer Physics Communications

April 1986

Lawrence Livermore National Laboratory

Over the past decade we have witnessed an evolution of scientific computers in which more and more concurrent or parallel arithmetic operations are allowed. The segmented pipeline arithmetic functional units, direct vectorization, indirect vectorization, multiprocessing, and finally multitasking represent stages of development of parallel computation. Algorithms for the solution of physics problems must be tailored, if possible, to the forms required for these various kinds of parallelism. Considerable experience in the adaptation of these methods to employ direct vectorization has been gained within the computational physics community over the last several years. Much of this know-how carries over to the newer forms of parallel computation. Indirect vectorization, available on some Cray X-MP models, on all Cray-2 computers, the ETA-10, and on some of the newer Japanese machines, allows one to vectorize data that is stored randomly in memory. Monte Carlo calculations have been greatly benefited by this development. In multitasking, one program, partitioned into tasks, may use one or more processors concurrently. The implementation of this technique is straightforward and will be shown to be applicable to more numerical techniques, including implicit ones, than were first thought to be the case. The new multi-processor computers are used optimally if one can continue to keep all of the processors busy most of the time. We do this by maintaining a time-sharing environment in which unitasking and multitasking jobs compete for the available processors in such a way as to minimize idling. Examples drawn from plasma equilibrium calculations, sparse matrix solvers, and particle simulations will be presented to illustrate some of these concepts.

1. INTRODUCTION

The history of parallel computing begins essentially at the beginning of the digital computer age, dating from the mid 1940's. The first ENIAC computer [1] was designed when digital circuitry was very slow. To obtain adequate speed it employed 23 functional units that were operated simultaneously from a single set of instructions that were literally hand coded using patch cords. Although arithmetic proceeded in parallel, the code was effectively sequential.

Rapid technological advances in computer speed and software difficulties with programming the parallel arithmetic units led to the abandonment of these early configurations in favor of ones for which the programming was more straightforward.

During the mid-1960's computers were developed that computed addresses of memory locations simultaneousely with the arithmetic of the application. This kind of concurrent operation operated behind the scenes, so to speak, and was of little concern to the programming physicist. Other operations, not of direct concern to the user, also became parallel. For example some computers could read and write to the memory by multiple data paths at the same time.

The development of vectorization came in two separate stages. About 1970, for example, the CDC 7600 was built with segmented arithmetic units that each consisted of approximately 7 separate stages through which operands are processed in an assembly line fashion; the last stage produces the final answer at the same time the earlier stages are processing operands through the preceding steps of the binary arithmetic. Yet, there was no hardware nor procedure for feeding a steady stream of operands into or results out of these segmented "pipeline" units. Typically, these segmented units were starved from the lack of available data and instructions. Thus they did not produce results at the desired rate of one answer per computer cycle.

The notion of a single instruction for combining operand vectors- the vector instruction- and the array of registers known as the vector register comprised the second development that permitted the segmented arithmetic units to run at a speed producing one result every computer cycle. Several vendors of scientific computers introduced such vector hardware and software during the mid-1970's; of these many of us are most familiar with the Cray-1 computer. A rough measure of the asymptotic speedup from vectorization is just the number of segments in the functional units on the order of 7.

With the advent of vectorization, the implementation of it could not be made automatic because certain kinds of recursions exist that could invalidate the vector mode. These could not (and can not) be detected by the compiler as there were (and are) dependencies on the data that cannot be forseen by the compiler. Therefore, the end user was given additional commands with which to control vectorization. New algorithms have been developed in response to vectorization. Certainly, programming styles have changed to exploit the faster speed. As one pays considerably less for vectorized calculations as compared to scalar ones there has been an economic incentive to use these techniques.

With this cursory history of parallel computing we now have set the stage for the discussion of several new developments that are beginning to have equally profound effects on the way in which we do computational physics. Three new items of hardware, that are relevant, have been incorporated in the Cray-2 computer: list

- A radically larger memory- typically 256M 64bit words (2 giga bytes).
- Indirect vectorization by gather and scatter operations.
- A multiprocessor architecture that has four CPU's sharing the large memory.

The sober realistic view is that the Cray-2 design was the result of compromise. One view contended that much larger memory was required to make the Cray-2 a more capable machine. A more traditional approach was to follow the historical sequence of increasing the speed and using fast memory that was expensive and thus could not be radically larger. To keep the Cray-2 affordable, the large memory option was chosen with the installation of cheap MOS (dynamic) memory. This has put a severe constraint on the hardware design which results in a machine that is not much more powerful, on a per-processor basis, than a Cray-1. Since it costs less than half a Cray-1 on a per processor basis, it is a much more economic machine. As we shall see, the memory, multitasking, and enhanced vectorization capabilities make it a very capable machine. Its main drawback, the MOS memory, and the consequent loss of chaining, were the result of the very high cost of fast static memory in recent years (compared to the MOS dynamic memory). Happily, static memory is now ten cents on the 1984 dollar. Future computers on the horizon will have large and fast static memory- as encouraged by the leading designers in the field.[17] The Cray-2 is supplying a critical link in this evolution by making supercomputers with very large memories available. As a result, many previously intractable physics applications are now affordable.

The outline of this paper is as follows: In the next two sections we shall attempt to describe the new kinds of parallelism that are allowed by the larger memory and by the indirect vectorization capability. Then in sections 4, 5, and 6 we'll discuss multiprocessing computers, unitasked multiprocessing, and multitasking. From some of our experiences and studies on the Cray-2, described in section 7, we then propose strategies for optimal use of the computer in section 8. In section 9 we briefly describe the CTSS multiprocessing system that we use and lastly, some concluding remarks will be found in section 10.

The related topics of microtasking, new languages with parallel syntax, automatic multitasking, and non-deterministic algorithms will not be discussed in this paper.

2. MEMORY PARALLELISM ??

At first one might not think of the much larger memory as being an aspect of parallel computing, but perhaps it can be regarded as a different form of parallelism in two signifcant ways In the first instance it obviates the need for the recomputation of intermediate quantities(that were previously done for the lack of available memory) thus shortening the number of cycles needed. One might regard the extra memory used as carrying these intermediate results in "parallel" with the other parts of the calculation. More importantly, the large memory allows the use of entirely different methods and their associated algorithms that were previously untenable because of their memory requirements. These other methods are chosen for their favorable convergence properties so that they require fewer computational cycles. Typically, higher order methods and more fully implicit algorithms are accomodated. Again, the memory holds the additional information in parallel with the actual computation

We illustrate this speedup effect of the memory by showing a comparison of matrix solvers used to advance a Fokker-Planck simulation of plasma transport. The iterative ADI solver of the CQL code[2] was replaced by a direct LU band solver. A recent calculation of the kinetics of runaway elections in a tokomak was redone with the new version. The implicit system to be solved has a matrix of order 15000. In the earlier version 13.52 minutes were spent doing matrix solves while in the new version just .53 minutes were used. Hence the speed increase was by a factor of 25.5. For the overall Fokker-Planck calculation a speedup factor of 15.1 was obtained. It should be realized that other problems might be better treated by an iterative method, but in this particular application the direct method produced the desired accuracy in much less time.

3. INDIRECT VECTORIZATION

Prior to the introduction of indirect vectorization, only data stored at regular constant intervals (or strides) in memory could be used in or written from vector operations. To implement the vectorization of randomly stored data, a list of addresses is needed since they cannot be computed from any regular pattern. Such a list vector is implemented in FORTRAN by using arrays with indirect indices such as A(I(J)) where the integer vector I(J) is this list vector. When A(I(J)) appears on the right side of a FORTRAN expression, the computer must gather the various A's from the indicated positions I(J) in memory and put them in a vector register for subsequent calculation. Alternatively, when A(I(J)) appears on the left side of a FORTRAN statement the computer must store- or scatter- the A's back to their positions I(J). The vector gather is straightforward, but the scatter can map several J values into the same memory location I, thus producing a vector conflict. One may remedy this problem by sorting into sub-vectors that each have one to one mappings. This feature has allowed Monte Carlo calculations, particularly plasma particle simulation codes, to be fully vectorized except for an inexpensive sorting phase. In Fig. 1 we show the improvement in performance obtained in the 3D particle code QN3D [3] for the portion of the calculation that interpolates grid quantities to the particle positions. This is a gather operation. In contrast, the interpolation of the particle quantities to the grid requires the scatter operation as well as the gather. Figure 2 shows the speed enhancement obtained when the scatter and gather operation was applied to properly sorted sub-vectors.

Another promising application of indirect vectorization is in the use of sparse matrix solvers. The use of indirect indices is a very convenient way to specify the sparsity pattern of the operator matrix. This also leads to very compact code compared to previous algorithms that did not employ this form. And in addition to this benefit, the compact form also vectorizes. The earlier forms of conjugate gradient solvers cannot be fully vectorized because several of the innermost loops are recursive. Our new method essentially builds a new loop inside this recursive one; the new loop vectorizes with a vector length basically equal to half the number of operator stencil points. In many applications this number is on the order of 10 to 100 and can lead to faster code. We have used this method to solve coupled partial differential equations in 3D in fully implicit form. As an illustration of this methodology we display portions of the FORTRAN coding for both the old and new cases. We show the part that implements the incomplete LU factorization of the sparse matrix. This kind of treatment is often found in preconditioned conjugate gradient algorithms such as ILUCG. Details of these methods and codes are available elsewhere. [4,5,6,7] One should bear in mind that the ILUCG3 code is restricted to the special case of just one PDE. The relevant section of code from the program ILUCG3 now follows:

```
c      do 10 i = 1, mn
c
c
c
c..This removed section had the calculation of b(i,l15), the diagonal.
c
c
c
c..The following code gets the off-diagonal factors of L and U
c
       in15 = i + n15
       b(i,l15) = b(i,l15) + b(i,l2 )*b(in15-n27,l27)
      . + b(i,l3 )*b(in15-n26,l26) + b(i,l5 )*b(in15-n24,l24)
      . + b(i,l6 )*b(in15-n23,l23) + b(i,l8 )*b(in15-n21,l21)
      . + b(i,l9 )*b(in15-n20,l20) + b(i,l11)*b(in15-n18,l18)
      . + b(i,l12)*b(in15-n17,l17)
c      in13 = i - n13 = in15
       b(in15,l13) = -b(i,l14)*(b(in15,l13)
      . + b(in15,l1 )*b(i-n26,l26) + b(in15,l2 )*b(i-n25,l25)
      . + b(in15,l4 )*b(i-n23,l23) + b(in15,l5 )*b(i-n22,l22)
      . + b(in15,l7 )*b(i-n20,l20) + b(in15,l8 )*b(i-n19,l19)
```

```
     . + b(in15,l10)*b(i-n17,l17) + b(in15,l11)*b(i-n16,l16))
       in16 = i + n16
       if(in16.gt.mn) go to 10
       b(i,l16) = b(i,l16) + b(i,l4 )*b(in16-n26,l26)
     . + b(i,l5 )*b(in16-n25,l25) + b(i,l7 )*b(in16-n23,l23)
     . + b(i,l8 )*b(in16-n22,l22) + b(i,l13)*b(in16-n17,l17)
c      in12 = i + n12 = in16
       b(in16,l12) = -b(i,l14)*(b(in16,l12)
     . + b(in16,l2 )*b(i-n24,l24) + b(in16,l3 )*b(i-n23,l23)
     . + b(in16,l5 )*b(i-n21,l21) + b(in16,l6 )*b(i-n20,l20)
     . + b(in16,l11)*b(i-n15,l15))
       in17 = i + n17
       if(in17.gt.mn) go to 10
       b(i,l17) = b(i,l17) + b(i,l4 )*b(in17-n27,l27)
     . + b(i,l5 )*b(in17-n26,l26) + b(i,l6 )*b(in17-n25,l25)
     . + b(i,l7 )*b(in17-n24,l24) + b(i,l8 )*b(in17-n23,l23)
     . + b(i,l9 )*b(in17-n22,l22) + b(i,l13)*b(in17-n18,l18)
c      in11 = i + n11 = in17
       b(in17,l11) = -b(i,l14)*(b(in17,l11)
     . + b(in17,l1 )*b(i-n24,l24) + b(in17,l2 )*b(i-n23,l23)
     . + b(in17,l3 )*b(i-n22,l22) + b(in17,l4 )*b(i-n21,l21)
     . + b(in17,l5 )*b(i-n20,l20) + b(in17,l6 )*b(i-n19,l19)
     . + b(in17,l10)*b(i-n15,l15))
       in18 = i + n18
       if(in18.gt.mn) go to 10
       b(i,l18) = b(i,l18) + b(i,l5 )*b(in18-n27,l27)
     . + b(i,l6 )*b(in18-n26,l26) + b(i,l8 )*b(in18-n24,l24)
     . + b(i,l9 )*b(in18-n23,l23)
c      in10 = i + n10 = in18
       b(in18,l10) = -b(i,l14)*(b(in18,l10)
     . + b(in18,l1 )*b(i-n23,l23) + b(in18,l2 )*b(i-n22,l22)
     . + b(in18,l4 )*b(i-n20,l20) + b(in18,l5 )*b(i-n19,l19))
       in19 = i + n19
       if(in19.gt.mn) go to 10
       b(i,l19) = b(i,l19) + b(i,l10)*b(in19-n23,l23)
     . + b(i,l11)*b(in19-n22,l22) + b(i,l13)*b(in19-n20,l20)
c      in9  = i + n9  = in19
       b(in19,l9 ) = -b(i,l14)*(b(in19,l9 ) + b(in19,l5 )*b(i-n18,l18)
     . + b(in19,l6 )*b(i-n17,l17) + b(in19,l8 )*b(i-n15,l15))
```

```
      in20 = i + n20
      if(in20.gt.mn) go to 10
      b(i,l20) = b(i,l20) + b(i,l10)*b(in20-n24,l24)
     . + b(i,l11)*b(in20-n23,l23) + b(i,l12)*b(in20-n22,l22)
     . + b(i,l13)*b(in20-n21,l21)
c     in8  = i + n8  = in20
      b(in20,l8 ) = -b(i,l14)*(b(in20,l8 ) + b(in20,l4 )*b(i-n18,l18)
     . + b(in20,l5 )*b(i-n17,l17) + b(in20,l6 )*b(i-n16,l16)
     . + b(in20,l7 )*b(i-n15,l15))
      in21 = i + n21
      if(in21.gt.mn) go to 10
      b(i,l21) = b(i,l21) + b(i,l11)*b(in21-n24,l24)
     . + b(i,l12)*b(in21-n23,l23)
c     in7  = i + n7  = in21
      b(in21,l7 ) = -b(i,l14)*(b(in21,l7 ) + b(in21,l4 )*b(i-n17,l17)
     . + b(in21,l5 )*b(i-n16,l16))
      in22 = i + n22
      if(in22.gt.mn) go to 10
      b(i,l22) = b(i,l22) + b(i,l10)*b(in22-n26,l26)
     . + b(i,l11)*b(in22-n25,l25) + b(i,l13)*b(in22-n23,l23)
c     in6  = i + n6  = in22
      b(in22,l6 ) = -b(i,l14)*(b(in22,l6 ) + b(in22,l2 )*b(i-n18,l18)
     . + b(in22,l3 )*b(i-n17,l17) + b(in22,l5 )*b(i-n15,l15))
      in23 = i + n23
      if(in23.gt.mn) go to 10
      b(i,l23) = b(i,l23) + b(i,l10)*b(in23-n27,l27)
     . + b(i,l11)*b(in23-n26,l26) + b(i,l12)*b(in23-n25,l25)
     . + b(i,l13)*b(in23-n24,l24)
c     in5  = i + n5  = in23
      b(in23,l5 ) = -b(i,l14)*(b(in23,l5 ) + b(in23,l1 )*b(i-n18,l18)
     . + b(in23,l2 )*b(i-n17,l17) + b(in23,l3 )*b(i-n16,l16)
     . + b(in23,l4 )*b(i-n15,l15))
      in24 = i + n24
      if(in24.gt.mn) go to 10
      b(i,l24) = b(i,l24) + b(i,l11)*b(in24-n27,l27)
     . + b(i,l12)*b(in24-n26,l26)
c     in4  = i + n4  = in24
      b(in24,l4 ) = -b(i,l14)*(b(in24,l4 ) + b(in24,l1 )*b(i-n17,l17)
     . + b(in24,l2 )*b(i-n16,l16))
```

```
      in25 = i + n25
      if(in25.gt.mn) go to 10
      b(i,125) = b(i,125) + b(i,113)*b(in25-n26,126)
c     in3  = i + n3  = in25
      b(in25,13 ) = -b(i,114)*(b(in25,13 ) + b(in25,12 )*b(i-n15,115))
      in26 = i + n26
      if(in26.gt.mn) go to 10
      b(i,126) = b(i,126) + b(i,113)*b(in26-n27,127)
c     in2  = i + n2  = in26
      b(in26,12 ) = -b(i,114)*(b(in26,12 ) + b(in26,11 )*b(i-n15,115))
      in27 = i + n27
      if(in27.gt.mn) go to 10
      b(i,127) = b(i,127)
c     in1  = i + n1  = in27
      b(in27,11 ) = -b(i,114)*b(in27,11 )
 10   continue
 20   continue
```

In our new program CPDES3(coupled partial differential equation solver for 3D) the above form becomes

```
      do 10 i=1,mn
c
c
c
c
c..This removed section had the calculation of b(i,ld), the diagonal.
c
c
c
c..The following code gets the off-diagonal factors of L and U.
c

      do lploop lp = ld+1, 2*ld-1
      if(i + np(lp).gt.mn) go to 10
      option assert(nohazard)
      do upsum ipp = 1, ippm(lp)
      if(i+npp(ipp,lp).ge.1) then
      b(i,lp) = b(i,lp) + b(i,lpp(ipp,lp))*b(i+npp(ipp,lp),lss(ipp,lp))
      endif
upsum continue
      lpd = 2*ld - lp
      if(lpd.lt.1) go to lploop
      option assert(nohazard)
      do dnsum ipp = 1, ippm(lp)
      if(i-npp(ipp,lpd).ge.1) then
      b(i-np(lpd),lpd) = b(i-np(lpd),lpd) +
     . b(i-npp(ipp,lpd),lpp(ipp,lpd))*b(i-np(lpd),lss(ipp,lpd))
      endif
dnsum continue
      b(i-np(lpd),lpd) = -b(i,ld)*b(i-np(lpd),lpd)
lploop continue
10    continue
```

The most obvious point here is the substantial reduction in tedious detailed programming when we employ the indirect indexing. Not only is the coding more tractable (and by inference more readable), it is also more general. Where the first block of coding is specifically restricted to the matrix sparsity pattern generated by a 3D 27 point stencil operating on a scalar field, the second block is much more general and can represent more complicated operators. For example, fully coupled Maxwell's equations have been treated by this method.[8] We must point out that this indirect indexing could have been used in earlier codes, but the lack of vector indirection would have produced very slowly running programs at much less than the nominal scalar speed.

As one might expect, indirect vectorization has more overhead than the more standard direct form. For those cases where the vectors are short ($<< 64$), indirect vectorization may not be faster than the scalar coding that does not employ indirect indices. In one comparison we ran ILUCG3 and CPDES3 for the same 27 point operator and obtained the disappointing result that the latter was ten times slower per conjugate gradient iteration. The vectors that actually appear in that CPDES3 calculation vary in length from 9 down to 1 with an average length of about 5. For vectors of these short lengths even direct vectorization (which is not applicable here) would not give much improvement and, as is seen here, indirect vectorization is worse.

Nevertheless, longer vectors occur in more complicated problems where the stencil operator may represent several coupled partial differential equations. Then the speed of the calculation improves. For three coupled equations, a Darwin form of Maxwell's equations being an example, the CPDES3 code improved to be only four times slower than the reference calculation. We expect compiler improvements and better written FORTRAN on our part will yield improvements on these preliminary findings.

From these rather pessimistic results of the comparisons of the matrix solvers we can still make some favorable inferences. The newer forms of the code are much more general and compact thus making the human user more efficient. In this regard the likelyhood of coding errors is greatly reduced. These newer forms do vectorize and thus partly offset the otherwize greater expense of the additional level of indirection. These forms extend our capability but do not always lead to better performance as they did in the example of the particle codes.

4. MULTIPROCESSING COMPUTERS

At the present time many diverse types of parallel processing computers are being developed. Many of these represent research and development efforts while a few are in commercial production. We shall restrict our attention to this latter group. We further narrow the scope of our discussion to just those computers with shared memory and whose CPU's retain vectorization.

We see two important reasons that users will prefer the multiprocessing computers over their predecessors: Cost and performance. Even without multitasking, the sharing of a much larger memory makes for more cost effective use of the memory (as is true for many shared resources) and additionally gives more capability, as we saw above where a memory intensive algorithm was used with concomittant improvement in performance. For the new forms of vectorization, further cost reductions and performance gains sometimes accrue. When multitasking is done performance is enhanced but, as we shall see, determining the cost benefits of multitasking is difficult. It is complicated by issues of human performance, throughput degradation, the charging algorithm, as well as others. All agree that codes using very large amounts of memory will be more cost efficient if they can be multitasked, but for more typical programs there is much discussion.

NMFECC Computers				
Attribute	Cray-1	Cray-1s	Cray XMP-22	Cray-2
Number of CPU's	1	1	2	4
Memory Size	1m	2m	2m	64m
Floating Hardware Segments	7	7	7	19
Clock Cycle	12.5ns	12.5ns	9.7ns	4.2ns
Memory Channels	1	1	3	1
Memory Recovery	4 clocks	4 clocks	4 clocks	51 clocks
Chaining	Yes	Yes	Yes	No
Vector Indirection	No	No	No	Yes
Asequential Memory Traffic	No	No	No	Yes

Table 1: Hardware Characteristics of the NMFECC Cray Computers

Cray Research has manufactured two lines of multiprocessors, the Cray X-MP and the Cray-2. Experiences, inferences, and strategies regarding these machines will be related in the following sections with emphasis on the Cray-2 (where we compute most often.) In Table 1 we give several of the important specifications of the various Cray computers. The Cray-2 is a complicated machine that differs significantly from the Cray-1 and Cray X-MP line. Its faster clock cycle is offset by its slower memory, its lack of chaining, its lack of overlapped memory traffic, and by its alternate cycle issue of instructions. This results in a machine with a per processor performance roughly equal to a Cray-1 if one does not exploit its new features.

In Fig. 3 we have plotted for several computers the maximum floating point arithmetic speed (for the entire computer) versus the full memory size. From it we note that the Cray-2 computers do not follow the historical trend that one can determine from the sequence of the CDC7600, the Cray-1, and the Cray X-MP. If we establish the scenario that certain physics codes merely refine their grids as they move from the smaller, slower computers to the more capable ones, we can establish how many MFLOPS are required to solve the same problem on the different machines in the same wall time. In the case of MHD calculations in 3D, an $n^{4/3}$ scaling results. Here n is the number of grid points. Likewise, certain Poisson solvers scale as $n\ln_2 n$. In the figure we have plotted these scaling trends as well. The historical sequence of the earlier productions (all of Seymour Cray) follow this scaling. The Cray-2 (and by rumor the Cray-3) specifications lie well outside this trend. Many existing physics codes (that scale as shown) will become very slow if one simply refines the grid to fill up the Cray-2 memory. Clearly, there is the incentive here to move to alternative algorithms that are more memory intensive and that use fewer CPU cycles.

5. UNITASKED MULTIPROCESSING

The primary mode of multiprocessing on the Cray multiprocessors is by the concurrent computation of independent unitasking codes on the separate processors. This is certainly true for the batch oriented COS system supplied by Cray and for the timesharing oriented CTSS operating system of Livermore. In the unitasked mode of operation, the multiprocessor computer can be regarded as a group of independent computers. The existence of the shared memory is not a complication for the user, but is managed by the system.

It should be emphasised that under normal conditions, unitasking yields the maximum throughtput of the multiprocessor because it avoids the overhead associated with multitasking. Maximum throughput is equivalent to minimizing the idle cycles and cycles devoted to additional overhead. However, when a unitasking program requires so much memory that no other codes will fit in the remaining memory space, then a situation is created in which the other processors are forced to idle with an associated reduction of the throughput. This suggests that very large programs should multitask to maintain high system efficiency.

For calculations that require very fast real time performance, unitasking may be too slow. Then regardless of program size, multitasking may help one achieve the desired speed.

Policy may discourage unitasking other codes as well; this can be done by constraining the charging algorithm to give unitaskers large surcharges. It is felt that future computers may be massively parallel and that the users must be encouraged to learn multitasking now rather than later. As will become more apparent below, this is in fact the policy at NMFECC.

6. MULTITASKING ESSENTIALS

We shall discuss a form of parallel programming and computing known as multitasking. We will restrict our presentation to the implementations used with the COS[9] and CTSS[10] operating systems for Cray computers. Other forms of multitasking exist and others will be invented but we shall ignore them here.

From a programming point of view multitasking means the partitioning of a program into tasks each of which is a chain of subroutines- a so-called caller-callee chain. If the logic and data structure of the code permits several tasks to run concurrently, then multitasking may be invoked without destruction of the data or logic. Thus it is desireable that tasks be independent.

From the view of the computer, or let's say the operating system, multitasking is accomplished by regarding tasks as the basic unit of work, replacing the program. For a unitasking job, there is no fundamental change. If a program has more than one task, the system may assign the multiple tasks to more than one processor.

When the user implements multitasking, other library routines that are required must be loaded and executed thus increasing the overhead expense of the calculation. This overhead tends to be constant for every invocation of multitasking so it is important that the amount of physics arithmetic be large in comparison. The ratio of the number of computer cycles of physics arithmetic to the number of overhead cycles is defined to be the granularity of a task. We shall see that keeping the granularity sufficiently large is often not difficult. Another concept relevant to multitasking is that of reentrancy. It means that there exist multiple program pointers, one from each CPU, working on your job. There may even be several pointers within the same subroutine simultaneously; this means that a subroutine can run in parallel with itself. While only one copy of the subroutine (in the form of its instructions) will exist, multiple copies of the subroutine's local data are generated, one for each task created. The values of local data are not preserved across calls because they are not unique and therefore no prescription for their ressurection is logical. Thus we see that a subroutine running in parallel with itself is not really cloned, but rather multiple storage areas are created for the local variables and multiple pointers are used from the multiple processors to the single copy of code. The user may specify that some local variables be saved. This causes them to be stored with the single copy of the subroutine code rather than with the task data and it further implies that saved local variables may change in strange ways if several concurrently running tasks are trying to modify them. Nevertheless, good FORTRAN programming practice will keep one from making too many mistakes of the kind alluded to here.

In the introduction we reviewed the parallelism introduced by segmented functional units and by vectorization. From a programming point of view one could say these are at the statement level and at the inner do loop level. Multitasking is at the subroutine level or higher. It is implemented by asking the system to create a task of a subroutine and execute it. Further, the calling program does not wait for the task to finish unless explicitely told to wait. In Table 2 we display some of the most common multitasking commands we use. The simplest of these, and the most often used, are those of the task manager library. To start a task running we use CALL TSKSTART. Its arguments include a task identifier array, the name of the subroutine to start, and the arguments passed to that subroutine. Typically, one would make several calls to TSKSTART to get several tasks started. The calling program itself is a task known as the root task. If nothing else is done, all of the tasks started, including the root task, are available to the computer system for parallel computation. The nature of the operating system, the load of competing time sharing jobs, and other factors imply that the exact phasing of the parallel tasks will rarely be the same from one run to the next. The ordering of the storing and fetching of critical data from the memory is likely to vary and errors of synchronization possible. In a typical calculation, the root task will need the results of the called tasks at some point. It is just before this point that a synchronization barrier must be used. A call to TSKWAIT does this; it makes the calling task wait for the completion of the task named by the TSKWAIT call.

So there you have it. Just two calls allow you to invoke multitasking. Probably 90% of the physics applications can be accomodated by this form without recourse to the more complicated synchronization calls we discuss

Multitasking Commands to MULTILIB					
Command	Effect	Facile	Overhead	Degrading?	Remarks
TSKSTART	Spawning	Yes	$.25ms$	No	Essential
TSKWAIT	Halts Caller	Yes	$.02ms$	No	Essential
LOCKASGN	Names Region	Maybe	$.02ms$	Maybe	Avoidable
LOCKON	Locks Region	Maybe	$.02ms$	Maybe	Avoidalbe
LOCKOFF	Unlocks	Maybe	$.02ms$	Maybe	Avoidable
EVASGN	Names Event	No	$.02ms$	Often	Avoidable
EVPOST	Satisfaction	No	$.02ms$	Often	Avoidable
EVCLEAR	Sets Condition	No	$.02ms$	Often	Avoidable
EVWAIT	Waits for Post	No	$.02ms$	Often	Avoidable

Table 2: Properties of the Most Common Multitasking Commands

below.

Sometimes, tasks are not fully independent of each other and pains must be taken to insure that the computation will proceed deterministically as intended by the user. We shall distinguish two kinds of synchronization refered to as locks and events.

Sometimes there exist regions of code that are critical in the sense that only one of these regions should be allowed to compute at a given time. Usually, it is the case that these several regions of code would modify the same data in memory and must have exclusive access to this data to prevent erroneous results. To protect these critical regions we surround every critical region of coding with CALL LOCKON(LID) and CALL LOCKOFF(LID) where LID is a lock identifier. The effect of these calls is to lock and unlock the critical regions of code on a first come first serve basis. When the region is locked, the other tasks wait when they reach the critical region. When it is unlocked, the first task that gets to it enters the region and locks it and then unlocks it when it is done. Sometimes one may avoid the use of locks by simply changing the data structure such that the critical regions are removed. It should be realized that when we say the "same data" we mean the very same memory locations. Different tasks may modify the same data array without difficulty so long as they don't hit the same locations.

In other instances, the tasks are mutually or singly dependent on the calculation of intermediate results. They must insure these results are existing before they can continue to run. When the various tasks have these dependencies one may use events to control the ordering of the computation. This is done principally by using the routines EVCLEAR(EID), EVPOST(EID), and EVWAIT(EID). The first of these says that an event, EID (the serial number name of the event), has not yet occured and that all tasks calling EVWAIT(EID) will stop until the event has occured. The event's occurence is invoked by the call to EVPOST(EID). In practice one can identify two principal uses for event management. The primary use is to generalize the TSKSTART and TSKWAIT combination to have lower overhead without any change in the logical flow of the program. Without getting into the details it allows one to keep tasks alive between their invocations so as to save the overhead of recreating the task at every recurrence of its use; sometimes this kind of use is refered to as barrier synchronization. The second use of it is required when a task has dependencies on other tasks that may be running concurrently. It should be apparent that concurrence implies that the exact ordering is not known. If there are points in the program where the ordering is important one must use events to control it. Needless to say, one can produce very complicated, hard to understand, and likely incorrect code with the event commands. Great care must be taken to insure that deterministic algorithms result and that they are the intended ones. As with locks, there are other remedies that sometimes would allow one to avoid events. For example, subdivision of tasks may result in a program that would not use events and that would only require TSKWAIT or a simpler barrier synchronization.

We wish to stress the point that locks and events reduce the parallelism of a code. The greater the portion of the code that they control, the closer the code will resemble a unitasking code in performance. Sometimes their use is unavoidable. Then very thorough testing must be done to insure determinism as well as faithfulness to the desired algorithm.

7. MULTITASKING EXPERIENCES

Within the computational physics group at NMFECC several multitasking codes have been built. Table 3 gives some few details of these codes. The first of these, VEPEC, is a rather large plasma equilibrium code that had already been highly vectorized. It is also implicit and nonlinear. We felt it offered us a good test bed on which we could experiment

Multitasking Codes from NMFECC						
Code	Purpose	Memory Size	Maximum Theoretical Overlap	Maximum Measured Overlap	Average Measured Overlap	Comments
ILUBCG2	Matrix Solver	9.2×10^6	1.81	1.79	1.67	Stiff
QN3D	Particle Code	1.2×10^7	4	3.7	2.8	Vectorized
SIMU	Turbulence	3.4×10^5	4	NA	NA	Easy
VEPEC	Equilibria	9.7×10^6	4	3.6	2.4	Implicit

Table 3: Overlap Results from Selected NMFECC Physics Codes

with multitasking to study conversion problems, to learn about the effects of granularity, and to see if implicit (recursive) structures have any hope of being multitasked. Some of our research has been in the area of sparse matrix solvers. We have built several preconditioned conjugate gradient solvers. Some of these employ the bi-conjugate gradient method that quite naturally splits into two independent tasks for about 80% of the calculation. Lastly, we have undertaken a project to design a major 3D plasma particle simulation code that is optimized for the Cray-2 in such a way that the code is highly vectorized (95%) and fully multitasked.

VEPEC STUDIES

Our first study with VEPEC looked at the effects of granularity. We chose a very simple loop in which two parts of the plasma current density are added to form the total current density. These three vectors are computed on a 3D domain which we unroll so that the calculation proceeds with just one spatial index. Vectorization is done over the spatial index and a separate task made for each three vector component. Next, we chose a somewhat larger granule of work in which we evaluate the multipole expansion on the exterior boundary surfaces for the determination of self-consistent boundary conditions. For a third case we looked for the largest granule existent in the code which turned out to be the evaluation of the plasma pressure tensor. It is a sum over several species's pressures that in some cases require numerical integration over velocity space coordinates. Several subroutines, 9 in all, are in the task chain we built. This calculation is fully vectorized along the z coordinate and the four tasks partition the x,y plane.

At this point we digress to show the coding details of the implementation of the multitasking of this large granule. The subroutine VUTP computes the plasma pressures along a group of axial grid lines. It does this by calling other vectorized routines that evaluate the various model calculations. In Table 4 we show the subroutines invoked in the large granule chain. We note the routines VSFUN, VTFUN, and VUFUN perform integrations over velocity space by generalized Gaussian quadrature techniques. Roughly 800 lines of FORTRAN comprise this granule. The way VUTP and its called routines are started is shown next.

```
      external vutp

      .if(nmtskon.ne.1) then
      do nkloop jk=1,nk
      call vutp(koff,ndbot,ndcomps,jk)
 nkloop continue
      .else
      call mtimer(0,jout)

      do nkloop jk = 1,nk
      tskbgrn(1,jk) = 3
      tskbgrn(2,jk) = 0
      tskbgrn(3,jk) = jk
      jkarg(jk) = jk
      call tskstart(tskbgrn(1,jk),vutp,koff,ndbot,ndcomps,jkarg(jk))
 nkloop continue

      do nkwait jk = 1,nk
      call tskwait(tskbgrn(1,jk))
 nkwait continue
      call mtimer(1,jout)
      .endif
```

In this section of code we note that MTIMER is a routine that returns timings including information about multitasking overlap. The parameter NMTSKON controls a compile time .if .else and .endif construct that has the effect of removing either the multitasked version or the unitasked version from the compiler input. The work is partitioned into NK parts and

Large Granule Subroutine Chain					
Routine	Chain Level	Caller	Purpose	Invocations	Fully Vectorized
PVAL	Parent	BOSS	Plasma Pressure	1	No
VUTP	Task	PVAL	Axial Vectors	4	No
VNPKERN	2nd	VUTP	MHD Pressures	4	Yes
VPOTMOD	3rd	VNPKERN	Potentials	4	Yes
VPASPRES	2nd	VUTP	Passing Ions	4	Yes
VTRPPRES	2nd	VUTP	Trapped Ions	4	Yes
VSLSPRES	2nd	VUTP	Sloshing Ions	4	Yes
VSFUN	3rd	...PRES	Velocity Integrals	> 4	Yes
VTFUN	3rd	....PRES	Velocity Integrals	> 4	Yes
VUFUN	3rd	....PRES	Velocity Integrals	> 4	Yes

Table 4: All of these Vectorized Routines Form the VUTP Task

the loop NKLOOP ranges over them. Within the unitasked loop sequential calls are made to VUTP until all the work is done. For the multitasking loop there is more coding but it is all straightforward. First the task identifiers are initialized. Next, the loop index is stored into the array jkarg(jk) because the tasks may want to refer to the index that created them rather than using its instantaneous value which changes rapidly. Then the call to TSKSTART sets VUTP running as a separate task; it is started repeatedly by subsequent calls without waiting for a return from VUTP until the NKLOOP is completed. Lastly, the loop NKWAIT contains the TSKWAIT call that causes the calling program, the root task, to wait until all of the tasks have completed. This granule is routinely multitasked in our production runs of the code. In a recent rather simple calculation that employed only two plasma species we have obtained overlap factors of about 2.8 meaning this granule executes 2.8 times faster than its unitasked counterpart.

In these studies of granularity we used both the Cray XMP and the Cray-2 because we expected dependencies on the machine architecture in the results. In Table 5 we show the number of floating point computer cycles that were required to complete each of the granule's work. Since VEPEC is non-linear it is solved by iteration. At every non-linear iteration, each of these granules is computed again. We looked at different spatial grids on the two computers such that the code would be of maximal permitted

FLOP and Granularity Comparisons		
Computer	Cray XMP-22	Cray-2
Multitasking Overhead Flop	6×10^4	6×10^4
JTOT:		
Flop	5.4×10^4	2.4×10^6
Granularity	0.90	40.0
EXPAND3:		
Flop	2.7×10^5	3.4×10^6
Granularity	4.50	56.7
VUTP:		
Flop	4.0×10^8	1.8×10^{10}
Granularity	6666.66	300000.0

Table 5: Granularity of Large Cray-2 Codes Versus the XMP

length on the respective machines. We did this because multitasking is most important at full or nearly full memory. The floating point cycle count must be compared with the task overhead which was determined to be approximately 6×10^4 cycles. One concludes that multitasking on the XMP will have a high overhead except in the large granule case. But on the Cray-2 all of the granules have relatively small overhead ratios. Obviously, what we dub as a small granule really has a granularity that depends on the dimensions of the arrays. Seemingly small loops have a tendency to become very large when in a large Cray-2 code. The big user should take some encouragement from this. Many small chunks of code, such as outer do-loops may be effectively multitasked in these large codes.

Of considerable interest to us was whether implicit algorithms could be conveniently multitasked? We are also addressing this issue by studying direct solvers (band solvers using Gaussian elimination), by developing multitaskable preconditioned conjugate gradient solvers, and by using cyclic reduction techniques. But within VEPEC we decided to modify the Douglas-Gunn (DG) algorithm [16], which is still a useful technique for problems that require non-linear iterations. Heretofore, the VEPEC code used a scalar version of this type of 3D ADI. Although this technique properly treats only 7pt spatial operator stencils. we have treated more general operators by using a mixed implicit-explicit technique. Now for the DG algorithm, we remind ourselves that it proceeds in three stages. Within each stage the algorithm is recursive in one of the spatial indices but independent in the transverse space. Thus we parallelize in these transverse spaces. Within each of them we divide the parallel work between vectorization and multitasking. One learns rather quickly to set the number of tasks equal to the number of processors and to vectorize within the tasks. Measurements of the performance show that vectorization alone yields a factor 4.1 speedup over the older scalar code. When multitasking and vectorization are used together we typically gain another factor of 2.5 speed up. The performance of multitasking depends critically on how the system schedules the processors to work on all the competing jobs and tasks available to it. For the various granules in the DG computation we measure overlap factors that vary from roughly 1.5 to about 3.6 with an average of 2.5 or so. We are encouraged by these results which were obtained for a version of the code that occupied about 10 million words of the Cray-2 about 1/7th the memory in house and only about 1/25th of the typical Cray-2 installation. As the memory requirement increases we would expect better overlap statistics. The main conclusion here is that an existing implicit method was successfully converted to multitasking with large performance gains.

BICONJUGATE GRADIENT SOLVERS

Recently, considerable interest has been shown in various sparse matrix solvers that use preconditioning and conjugate gradient techniques. These techniques are actually exact after a finite number of algorithmic cycles for the ideal case of an infinite precision machine, but in the real world they can be regarded as iterative. This ideal number of cycles is just the number of distinct eigenvalues of the matrix which is less than or equal to the order (the number of linear equations) of the matrix. When properly preconditioned, convergence is very fast. In fact, precision to machine roundoff is typically obtained in many fewer cycles than for the ideal case.

There are, of course, alternative methods such as direct solvers that are better in some circumstances. In comparison to the conjugate gradient methods, the direct solvers are better for small order matrix systems. For sufficiently large systems, the iterative conjugate gradient methods are faster. The exact size at which the various alternative methods perform equally varies according to the system software and hardware being used and also according to how clever one is in coding the algorithms. On the

Cray-1 and Cray X-MP computers, the lack of memory often precluded the use of the direct method even when the direct method would have been faster. For the Cray-2 computers, as we saw in a case of a Fokker-Planck code, a direct solver was much faster than an iterative one. But with matrix systems of order about 30000 or greater, the conjugate gradient methods win. If one does not need machine roundoff accuracy, then the CG methods take fewer cycles and become even more competitive.

In many plasma physics problems, one obtains a non-symmetric matrix equation to solve. Incomplete factorization techniques have been used to precondition their algorithms, such as ILUCG[5]. These methods conventionally have been developed to solve the normal form of the preconditioned problem. By normal form we mean that the system has been symmetrized by multiplying the matrix system by the transpose. This has the undesireable effect of squaring the condition number (or the spread of eigenvalues) of the original system. Mikic and Morse have used an alternate algorithm that is a generalization of the conjugate gradient algorithm that does not rely on the normal form.[11] This method, the bi-conjugate gradient algorithm[12], typically converges in far fewer iterations than the ILUCG method.

It has been noted that the recursive portion of the BCG method was split in two independent procedures and that these represented about 80% of the work per bi-conjugate gradient iteration. A modified form of BCG that equallizes the work in these two procedures has been developed.[13] Multitasking was then applied to the resulting method. The class of matrices treated were the same as those of the ILUCG2 studies of Shestakov et. al.[5] which pertained to the 2D non-symmetric 9point operator stencil appropriate for partial differential equations on a scalar field.

The new code ILUBCG2 was then compared to ILUCG2 for the same test problems and was found to converge about twice as fast. Both methods have the same amount of arithmetic work per iteration, so the ILUBCG2 code was faster even when run in unitasking mode. Studies have been done multitasking ILUBCG2 on both the Cray X-MP22 and the Cray-2 in our center. We show some of the results in Table 6. Of particular interest is the CPU overlap result. For this code, of approximately 9 million words in size, the overlap of 1.67 is quite close to the theoretical maximum overlap of 1.80. One obvious feature of the results is that larger codes tend to get better overlap. It should be realized that these overlap results, which really reflect wall time performance, are very sensitive to the system scheduler properties. The scheduler in use on the Cray-2 at NMFECC

COMPARISON: UNI- VS MULTITASKING							
Total CPU in Loop	CPU per iteration	CPU per Task	Max Possible Overlap	Time-Sharing Overlap	CPU in Solver	Overlap in Solver	Wall Clock time
CRAY-2 using 2 processors							
17.58	1.46	.65	1.80	1.67	18.77	1.60	16.82
17.76	1.48	.66	1.80	1.65	18.94	1.58	18.68
17.56	1.46	.65	1.81	1.66	18.74	1.59	15.82
17.51	1.46	.65	1.80	1.68	18.69	1.60	16.96
Averages:							
17.60	1.46	.65	1.80	1.67	18.79	1.59	17.07
CRAY-2 Unitasking							
17.84	1.49				19.04		23.37
17.17	1.48				18.90		21.44
17.70	1.47				18.87		22.42
17.62	1.47				18.80		21.63
Averages:							
17.72	1.48				18.90		22.22

Table 6: Comparsion of multitasking and unitasking modes for the CRAY-2.

has been modified to give good performance to multitasking jobs without degradation of the system throughput. For sufficiently large codes, it is possible to actually use computer time at a rate faster than real time if one can obtain sufficient overlap, even in our timesharing environment. In Table 7 a comparison of unitasking and multitasking of ILUBCG2 shows that in one case 19 seconds of CPU time were delivered in 14 seconds. This should be conclusive evidence of real parallel computation.

Recently, we have incorporated the multitasked BCG method into our coupled partial differential equation solver.[14] Thus the scalar multitasked backsolves of ILUBCG2 generalize to have indirect vectorization instead of the recursive scalar coding used previously.

FULLY PARALLELIZED PARTICLE CODES

In the foregoing discussion we described conversions to multitasking and also the construction of a multitasking matrix solver, but we have not

COMPARISON: UNI- VS MULTITASKING							
Dimension 361 × 901				Dimension 241 × 601			
	CPU in Solver	Overlap in Solver	Wall Clock Time		CPU in Solver	Overlap in Solver	Wall Clock time
$p = 3$	18.78	1.64	14.13	$p = 3$	8.27	1.62	7.69
	18.91	1.63	14.20		8.2[illegible]	1.64	6.61
	18.88	1.60	14.10		8.20	1.64	6.19
	18.95	1.64	13.40		8.22	1.64	6.24
	18.90	1.65	13.49		8.22	1.63	8.04
Ave:	18.88	1.63	13.86	Ave:	8.22	1.63	6.95
Unitask	19.10		22.41	Unitask	8.18		12.03
	19.06		22.35		8.2[illegible]		11.49
	18.93		21.75		8.17		10.84
Ave:	19.03		22.17	Ave:	8.18		11.46

Table 7: Wall-clock time speedup using three processors on the CRAY-2.

considered the design of a full physics code in this Cray-2 multiprocessor environment. We decided to build a particle simulation code for plasma physics studies of compact toroidal confinement devices. Design constraints on the code were that it must be fully vectorized and fully multitaskable.

The code QN3D (Quasi-neutral 3D) is intended to study non-linear growth of the tilting mode of field-reversed configurations (FRC) of interest not only in the fusion program but in other plasma physics applications. It uses rather standard methods of particle simulation including linear weighting and the Boris particle pushing scheme[15]. We already discussed the use of the vector indirection in the interpolation routines in an earlier section. To multitask this code is trivial. The parallel work is simply partitioned among four tasks which in turn are fully vectorized.

We intend to use QN3D on problems where the memory requirement will be as large as allowed by the system. A very large discount will be obtained for multitasking and the full use of all the processors will prevent the possibility of forced idling of CPUs due to lack of memory for other codes.

8. STRATEGIES FOR PARALLEL COMPUTATION

Based on these experiences and with knowledge of the software and hardware system parameters, one may make estimates about computer performance and the associated costs. From this one can evolve a strategy for computing on multiprocessors such as the Cray-2.

We shall group our recommendations into two classes. The first class will consist of those practices that will likely improve system throughput (or at least maintain it) and/or will give better performance against the wall clock. In a second group we shall consider the effects of the charging algorithms and suggest further strategies.

Our first recommendation is to review the algorithms of a code to see if they can be replaced by alternate ones that use more memory but fewer arithmetic cycles. This way one may take adavantage of the extremely large memory. One's codes will finish sooner thus increasing system throughput while getting better performance. Next, one should look at what can be vectorized. With the vector indirection capability, many constructs may be vectorized that could not before. Vectorization yields both faster code and also benefits system performance. Finally, one should consider multitasking. The system at NMFECC is set up to give multitasking codes, even small ones, much better performance against the wall clock than can be obtained unitasking. But unlike the use of the memory and vectorization capabilities, multitasking does not increase system throughput for small or medium sized codes. One could term this a "zero sum game." As we have indicated, very large codes are an exception to this rule because they abuse other CPU's.

There are other reasons to multitask. Money. It is considered likely that future supercomputers will be massively parallel. The time may come when the computers will have more processors than active users in which case multitasking will be essential just to keep the machine busy. Also, memory sizes may not continue to grow at the same pace as the number of processors thus providing further encouragement for multitasking.

There is a certain resistance or inertia among users to the new methodology. We at NMFECC are trying to provide further incentives for the users to multitask so that they will be ready for the next generation's machinery. This policy is fostered by a charging algorithm that charges separately for memory and CPU time. Since the physical hardware is, costwise, about 2/3 memory and 1/3 CPU's, the charging alogorithm is designed to reflect this ratio in its formulation. The amount of CPU time used by a code will not change much from unitasking to multitasking as it represents a sum over the separate CPU's. However, the charge for memory depends on the product of the memory residency time by the memory size. Multitasking can radically reduce the memory residency charge up to a factor 4 on the Cray-2. The charging formula presently in use is given by

$$V = (1 - am)t_c + bmt_p$$

where $a = 2.0 \times 10^{-8}$, $b = 1.1 \times 10^{-7}$, and m is in words of memory. We have defined $\mathcal{M}$ as the ratio of CPU time (t_c) to the time of having at least one processor (t_p). Thus we have $t_p = t_c/\mathcal{M}$. Upon substitution and division by tc we obtain the charge per unit CPU time

$$R = 1 - 2 \times 10^{-8} m + 1.1 \times 10^{-7} m/\mathcal{M}$$

Using these formulae, in Fig. 4 we show how the cost per unit CPU time varies as a function of the memory charge. A large penalty is exacted for unitasking large memory codes. At the present user memory limit of 4×10^7 words, there is a 4.6 surcharge factor for unitasking. In contrast, a fully multitasking code sees a much smaller charge increase with a factor of about 1.3. For purposes of comparison we show what the charging formula could be if we were not providing a large incentive for multitasking; this is shown by the dashed lines. Even here we have a memory charge that is based on the idea that very large unitasking jobs would abuse other CPU's and should pay for the forced idling thus engendered.

As one might guess there are many viewpoints about the charging formula. Its derivation could be based on cost analysis, market analysis, policy constraints, industrial practice, or perhaps on a combination of these. In the multiproccessing environment, there are more degrees of freedom that further complicates the charging strategies as compared to the unitasking situation. As the figure indicates, the charging algorithm is preliminary and likely to be altered as it is better understood.

9. THE CTSS PRESCRIPTION

The Cray TimeSharing System (CTSS) was developed by NMFECC at Livermore to provide a time sharing operating system for the Cray-1 computers that provides a flexible, interactive and good debugging environment for the user. It was adapted from the earlier LTSS system (Livermore

Time Sharing System) that had been used on CDC computers. With the arrival of the multiprocessing computers, the Cray X-MP and the Cray-2, CTSS has been extended to them. The basic principle of interactivity is still adhered to and for good reason. It allows for the best use of human time and it allows debugging in the real computer environment. Features have been added to it to allow and indeed foster multitasking and multiprocessing in general.

Before multitasking was included the CTSS employed a scheduler algorithm that allowed the programs resident in memory to take turns running on the CPU's. It kept all of the processors busy most of the time so long as there were codes in memory available to run. With multitasking it looks at the tasks belonging to programs and schedules them to run. Multitasking programs are in the same queue with the unitasking ones. When a multitasking program comes to take its turn, it really gets several turns in sequence (with some exceptions) that allow it to get several processors started in rapid succession.

10. CONCLUSION

Multiprocessing scientific computers manufactured by Cray have been available to us for somewhat more that one year. We have explored some of the features of these new machines with most of our efforts being on the Cray-2. The large memory, the enhanced vectorization capability, and the multiprocessor architecture have been studied. Many of us are aware of certain deficiencies of the Cray-2. Many of these, on close inspection, represent trade-offs that were made to make the machine affordable. Cheap MOS memory was used that has a relatively poor recovery time. To maximise the fetch and store rate to memory required changes in the CPU architecture that, for example, removed the chaining feature of the earlier Cray computers. What we have learned is that, inspite of certain difficulties, the Cray-2 is a very good performer. This should be particularly evident if one assesses the capital cost per installed MFLOP.

We have converted, revised, and built algorithms to test the performance of the Cray-2 with the CTSS multitasking operating system. We have gained confidence that not only is multitasking easy to program, but also we find that many more numerical algorithms may use it than was earlier thought possible. It seems clear that future computers will be multiprocessors and it seems likely that they will be massively parallel so that eventually multitasking may well be a necessity for the average code.

11. ACKNOWLEDGEMENTS

Many colleagues from the systems programming staff at NMFECC have been very instrumental in helping us learn about the new hardware and software and have helped us by making systems changes that have improved the performance of parallelized codes. We thank Lawrence A. Diegel, Keith J. Fitzgerald, Kirby W. Fong, Timothy G. Pierce, and F. David Storch in this regard. For special thanks we note that Larry G. Berdahl played a signal role in getting us adapted to the Cray-2 and its software. This work benefited from the encouragement of John Killeen and Arthur A. Mirin without which little of this work would have been attempted. This work was performed under the auspices of the U.S.D.O.E. by the Lawrence Livermore National Laboratory under contract W-7405-ENG-48.

References

[1] Jack Worlton, "A Philosophy of Supercomputing," Los Alamos Scientific Laboratory Report LA-8849-MS, Los Alamos, New Mexico, 87545, USA.

[2] G. D. Kerbel and M. G. McCoy, Phys. Fluids 28, p. 3629.

[3] E. J. Horowitz, "Vectorizing the Interpolation Routines of Particle-in-cell Codes," accepted for publication in J. Comput. Phys. (1986)

[4] D. V. Anderson and A. I. Shestakov, Comput. Phys. Commun., **30**, (1983), pp. 37-42

[5] A. I. Shestakov and D. V. Anderson, Comput. Phys. Commun., u **30** (1983), pp. 31-36

[6] D. V. Anderson, Comput. Phys. Commun., **30**, (1983), pp. 51-57

[7] D. V. Anderson, Comput. Phys. Commun.. **30**, (1983), pp. 43-49

[8] D. V. Anderson, E. J. Horowitz, A. E. Koniges, and D. E. Shumaker, "Fully Implicit Solution of Maxwell's Equations in Three Dimensions by Preconditioned Conjugate Gradient Methods with an Application to Reversed Field Configurations," Proc. 13th European Conference on Controlled Fusion and Plasma Heating, Schliersee, FRG, April 1986.

[9] Cray Multitasking Users Guide, SN-0222, CRI Publications Department Mendota Heights, MN 55120, USA

[10] Kirby W. Fong, NMFECC Buffer Articles, (On new linkages and multitasking), July 1984 - April 1985.

[11] Z. Mikic and E. C. Morse, "The Use of a Preconditioned Biconjugate Gradient Method for Hybrid Plasma Stability Analysis," to appear in J. Comput. Phys. (1986)

[12] D. A. H. Jacobs, in "Sparse Matrices and their Uses," (I. S. Duff, ed.), Academic Press Inc., New York, (1981), pp. 191-222

[13] A. E. Koniges and D. V. Anderson, "ILUBCG2: A Preconditioned Biconjugate Gradient Routine for the Solution of a Linear Asymmetric

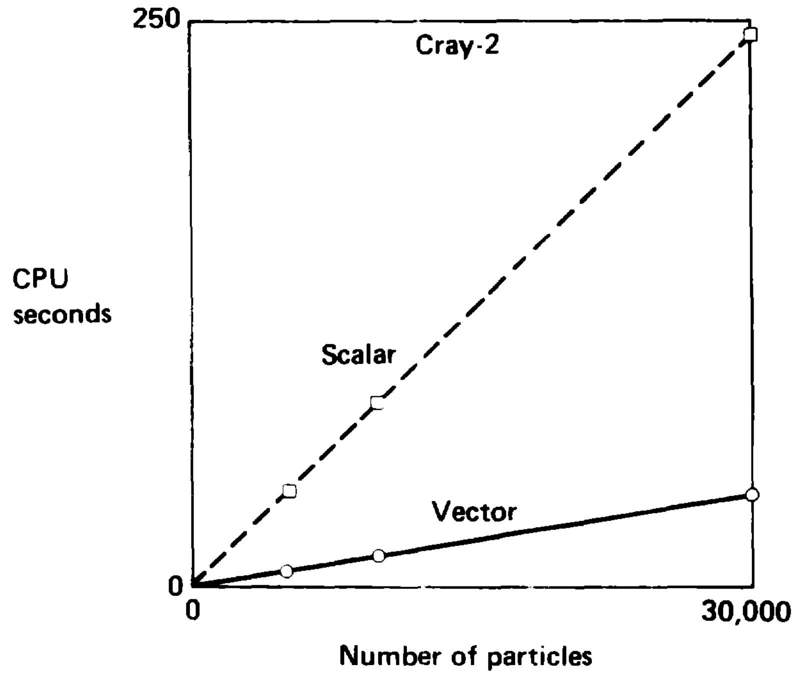
Matrix Equation Arising from a 9-Point Discretization, to be submitted to Comput. Phys. Commun.; also in Lawrence Livermore National Laboratory Report UCRL- 93616 (1986)

[14] D. V. Anderson, E. J. Horowitz, A. E. Koniges, and D. E. Shumaker, "A Fully Implicit Field Solver for 3D Particle Simulation Codes," Proc. 1986 Sherwood Theory Conference, New York City, (1986).

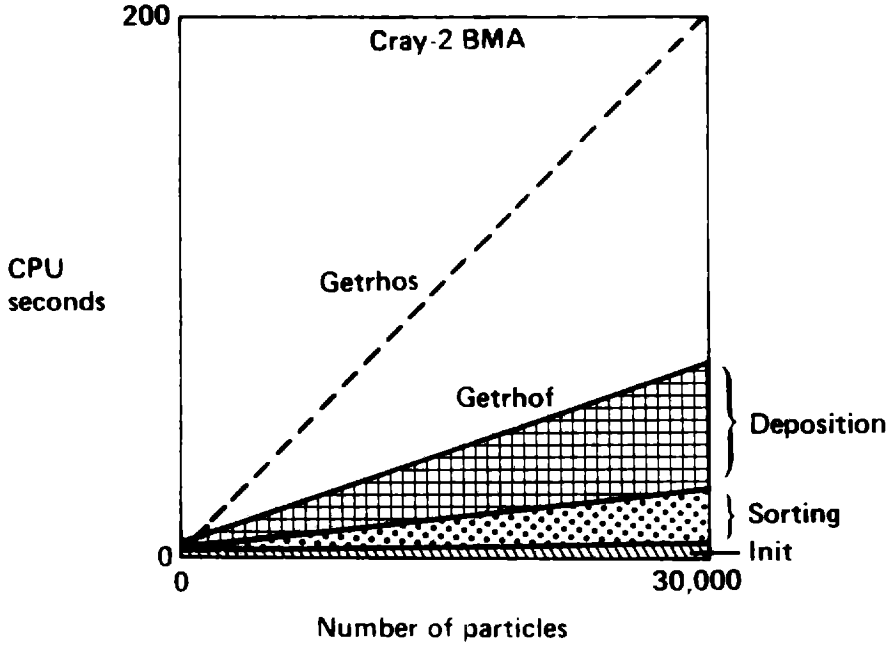
[15] J. P. Boris, "Relativistic Plasma Simulation- Optimisation of a Hybrid Code," Proc. 4th Conf. on Numer. Simulation of Plasmas, Office of Naval Research, Arlington, Virginia USA, pp. 3-67

[16] J. Douglas and J. Gunn, Numer. Math. **6**, (1964), p. 428

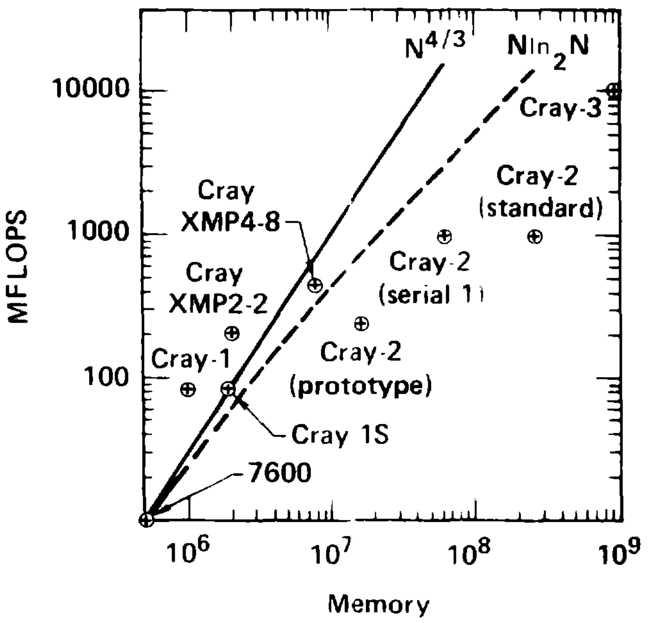
[17] L. G. Berdahl, private communication.



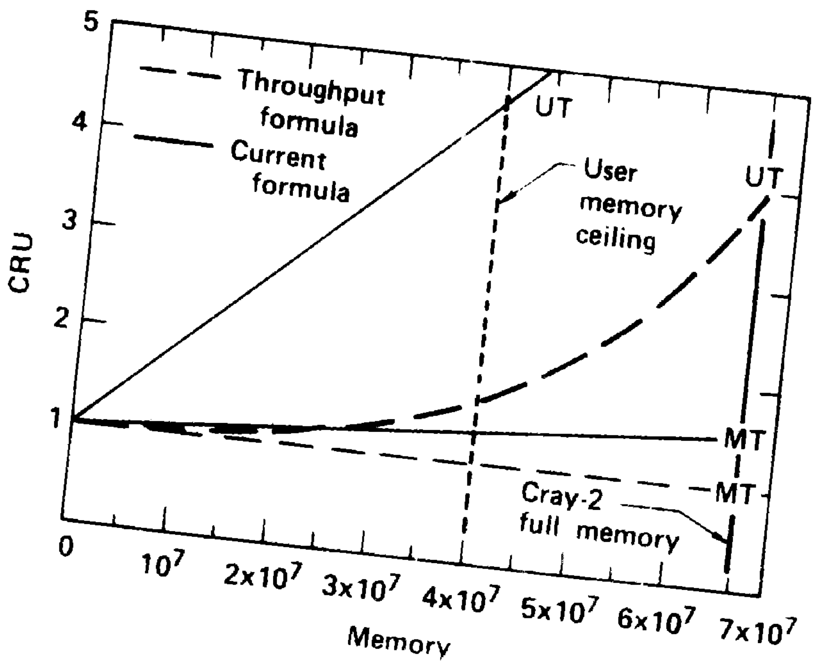
250
Cray-2
CPU seconds
Scalar
Vector
0
0
30,000
Number of particles



200
Cray-2 BMA
CPU seconds
Getrhos
Getrhof
Deposition
Sorting
Init
0
0
30,000
Number of particles



MFLOPS
10000
1000
100
Memory
10^6
10^7
10^8
10^9
N^4/3
Nln_2N
Cray-3
Cray-2 (standard)
Cray-2 (serial 1)
Cray-2 (prototype)
Cray XMP4-8
Cray XMP2-2
Cray-1
Cray 1S
7600



5
4
3
2
1
0
CRU
Throughput formula
Current formula
UT
User memory ceiling
UT
MT
MT
Cray-2 full memory
10^7
$2x10^7$
$3x10^7$
$4x10^7$
$5x10^7$
$6x10^7$
$7x10^7$
Memory